CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Archivio istituzionale della ricerca - Università di Trieste

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 60 (2015) 573 – 582
Procedia Computer Science

19th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

Edge-based mining of frequent subgraphs from graph streams

Alfredo Cuzzocrea[a], Zhao Han[b], Fan Jiang[b], Carson K. Leung[b,*], Hao Zhang[b]

[a]*Dept. of Engineering and Architecture (DIA), University of Trieste & ICAR-CNR, Via A. Valerio 6/1, 34127 Trieste (TS), Italy*
[b]*Department of Computer Science, University of Manitoba, Winnipeg, MB, R3T 2N2, Canada*

Abstract

In the current era of Big data, high volumes of valuable data can be generated at a high velocity from high-varieties of data sources in various real-life applications ranging from sensor networks to social networks, from bio-informatics to chemical informatics. In addition, Big data are also available in business, education, engineering, finance, healthcare, scientific, telecommunication, and transportation domains. A collection of these data can be viewed as a big dynamic graph structure. Embedded in them are implicit, previously unknown, and potentially useful knowledge. Consequently, efficient knowledge discovery algorithms for mining frequent subgraphs from these dynamic streaming graph structured data are in demand. On the one hand, some existing algorithms discover collections of frequently co-occurring edges, which may be disjoint. On the other hand, some other existing algorithms discover frequent subgraphs by requiring very large memory space. With high volumes of Big data, available memory space may be limited. To discover collections of frequently co-occurring *connected* edges, we present in this paper two efficient algorithms that require *small* memory space. Evaluation results show the efficiency of our edge-based algorithms in mining frequent subgraphs from graph streams.

© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of KES International
Keywords: Knowledge discovery and data mining; frequent patterns; frequent subgraphs; graph structured data; data streams

1. Introduction and related works

With the automation of measurements and data collection, together with an increasing development and usage of a large number of sensors, high volumes of valuable data have been produced at high velocity from a high variety of data sources in different application areas—such as bio-informatics, chemical informatics, e-commerce, education, engineering, finance, healthcare, science, sports and telecommunications[22,30]—in the current era of Big data[17,21]. Mostly due to their high volumes, the quality and accuracy of data depend on their veracity (i.e., uncertainty of the data[18,24,25,26]). These advances in technology have led to streams of semantic web, sensor network, social network, and road network data[9,12,15,29]. These kinds of data share in common the property of being modeled in terms of graph-structured data[10,28] so that *graph streams* are generated. Embedded in these data are implicit, previously unknown, and potentially useful knowledge. In order to be able to make sense of streaming data[7,8,27,31], stream mining algorithms

* Corresponding author.
E-mail address: kleung@cs.umanitoba.ca (C.K. Leung)

1877-0509 © 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of KES International
doi:10.1016/j.procs.2015.08.184

are needed. When comparing with mining from traditional *static* databases, mining from *dynamic* data streams[11] is more challenging due to the following properties of data streams:

1. *Data streams are continuous and unbounded.* To find frequent patterns from streams, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Hence, we need some data structures to capture the important contents of the streams (e.g., recent data—because users are usually more interested in recent data than older ones).
2. *Streaming data are not necessarily uniformly distributed; their distributions are usually changing with time.* A currently infrequent pattern may become frequent in the future, and vice versa. So, we have to be careful not to prune infrequent patterns too early; otherwise, we may not be able to get complete information such as frequencies of certain patterns (as it is impossible to retract those pruned patterns).

These two properties play an important role in the mining of data streams in general. They play a more challenging role in the mining of a specific class of streaming data—namely, *streams of graph structured data*.

Over the past decade, both approximate and exact algorithms have been proposed to mine frequent patterns from *data* streams. For instance, *approximate* algorithms (e.g., FP-streaming[13], TUF-streaming[20]) focus mostly on efficiency. However, due to approximate procedures, these algorithms may find some infrequent patterns or miss frequency information of some frequent patterns (i.e., some false positives or negatives). An *exact* algorithm mines only truly frequent patterns (i.e., no false positives and no false negatives) by (i) constructing a Data Stream Tree (DSTree)[23] to capture contents of the streaming data and then (ii) recursively building FP-trees for projected databases based on the information extracted from the DSTree.

In recent years, several solutions have been proposed for mining *graph* streams. For instance, Aggarwal et al.[1] studied the research problem of mining dense patterns in graph streams, and they proposed probabilistic algorithms for determining such structural patterns effectively and efficiently. Bifet et al.[2] mined frequent closed graphs on evolving data streams. Their three innovative algorithms work on coresets of closed subgraphs, compressed representations of graph sets, and maintain such sets in a batch-incremental manner. Moreover, Valari et al.[32] discovered top-k dense subgraphs in dynamic graph collections by means of both exact and approximate algorithms. Furthermore, Chi et al.[6] proposed a fast graph stream classification algorithm that uses discriminative clique hashing (DICH), which can be applicable for OLAP analysis over evolving complex networks. We[3] previously mined frequent patterns—in the form of collections of frequently co-occurring edges—from dense graph streams. Specifically, our previous solution finds collections of frequently co-occurring edges, which include *connected* as well as *disjoint* edges. In many real-life situations (e.g., social or business applications[4,16,19]), it is desirable to obtain collections of frequent disjoint edges so as to help the discovery of the missing links (e.g., connect two or more disjoint groups of social entities sharing common research or business interests).

In some other situations, it is more efficient to find only the collections of frequent *connected* edges. Hence, in this paper, we present two algorithms that find collections of frequently co-occurring *connected* edges from streaming graph structured data:

1. Our first algorithm is an indirect 2-step one that first discovers all frequent edges and then prunes irrelevant (disjoint) edges at a post-processing step, whereas
2. our second algorithm is a direct 1-step one that pushes the pruning step early in the mining process for discovering all frequent connected edges.

Consequently, regardless which of the two edge-based algorithms was applied, only relevant patterns (i.e., frequent *connected* subgraphs) are returned to users. Moreover, as high volumes of streaming graph structured data can be generated at a high velocity, data may be too big to fit into memory. Both algorithms were designed in such a way that they use *limited memory* in the efficient mining of frequent subgraphs from graph streams.

The remainder of this paper is organized as follows. The next section gives background. Section 3 presents our first algorithm, which builds an on-disk data structure to capture and maintain relevant streaming graph structured data, recursively discovers collections of frequent edges, and then prunes those disjoint edges at a post-processing step. Section 4 presents our second algorithm, which pushes the pruning step early in the mining process. Evaluation results and conclusions are given in Sections 5 and 6, respectively.

2. Background

In this section, we provide background on frequent pattern mining from streams, with a focus on stream mining with (i) a global *DSTree*, (ii) a global *DSTable*, and (iii) a global *DSMatrix*. These three data structures were designed to serve as global structures for capturing important contents of batches of streaming transaction data within the current sliding window. With these global structures, local structures (e.g., local FP-trees) can be built from which frequent patterns can be mined.

2.1. Stream mining with a data stream tree (DSTree)

An exact algorithm mines frequent patterns from streaming data by first constructing a *Data Stream Tree* (*DSTree*)[23], which is then used as a global tree for recursive generation of smaller FP-trees (as local trees) for projected databases. Due to the dynamic nature of data streams, frequencies of items are continuously affected by the insertion of new batches (and the removal of old batches) of transactions. Arranging items in frequency-dependent order may lead to swapping—which, in turn, can cause merging and splitting—of tree nodes when frequencies change. Hence, in the DSTree, transaction items are arranged according to some canonical order (e.g., alphabetical order), which can be specified by the user prior to the tree construction or mining process. Consequently, the DSTree can be constructed using only a single scan of the streaming data. Note that the DSTree is designed for processing streams within a sliding window. For a window size of w batches, each tree node keeps (i) an item and (ii) a list of w frequency values (instead of a single frequency count in each node as in the FP-tree for frequent pattern mining from *static* databases). Each entry in this list captures the frequency of an item in each batch of dynamic streams in the current window. By so doing, when the window slides (i.e., when new batches are inserted and old batches are deleted), frequency information can be updated easily. Consequently, the resulting DSTree preserves the usual tree properties that (i) the total frequency (i.e., sum of w frequency values) of any node is at least as high as the sum of total frequencies of its children and (ii) the ordering of items is unaffected by the continuous changes in item frequencies.

Such a global DSTree is always kept up-to-date when the window slides. The actual mining process is "delayed" until it is needed. To start mining, the mining algorithm first traverses relevant tree paths upwards and sums the frequency values of each list in a node representing an item (or a set of items)—to obtain its frequency in the current sliding window—for forming an appropriate projected database. Afterwards, the algorithm constructs a local FP-tree for the projected database of each of these frequent patterns of only 1 item (i.e., 1-itemset) such as an $\{x\}$-projected database (in a similar fashion as in the FP-growth algorithm for mining static data[14]). Thereafter, the algorithm recursively forms subsequent FP-trees for projected databases of frequent k-itemsets where $k \geq 2$ (e.g., $\{x, y\}$-projected database, $\{x, z\}$-projected database, etc.) by traversing paths in these FP-trees. As a result, the algorithm finds all frequent patterns. As items are consistently arranged according to some canonical order, the algorithm guarantees the inclusion of all *frequent* items using just upward traversals. Moreover, there is also no worry about possible omission or double-counting of items during the mining process. Furthermore, as the DSTree is always kept up-to-date, all frequent patterns—which are embedded in batches within the current sliding window—can be found effectively.

Note that the DSTree mainly relies on the assumption—usually made for many tree-based algorithms[14]—that all trees (i.e., the global tree together with subsequent FP-trees) fit into the memory. For example, when mining frequent patterns from the $\{x, y, z\}$-projected database, the global tree and three subsequent local FP-trees (for the $\{x\}$-, $\{x, y\}$- and $\{x, y, z\}$-projected databases) are all assumed to fit into memory. However, there are situations (e.g., for streaming graph structured data) where the memory is so limited that not all these trees can fit into memory.

2.2. Stream mining with a data stream table (DSTable)

To deal with situations where the memory is so limited that not all these trees can fit into memory, the *Data Stream Table* (*DSTable*)[5] was proposed. The DSTable is a two-dimensional table that captures on the disk the contents of transactions in all batches within the current sliding window. Each row of the DSTable represents a domain item. Like the DSTree, items in the DSTable are arranged according to some canonical order (e.g., alphabetical order), which can be specified by the user prior to the construction of the DSTable. As such, table construction requires only a single scan of the stream. Each entry in the resulting DSTable is a pointer that points to the location of the table entry (i.e., which row and which column) for the "next" item in the same transaction. In addition, the DSTable also keeps

w boundary values (to represent the boundary between w batches in the current sliding window) for each item. By doing so, when the window slides, transactions in the old batch can be easily identified for removal and transactions in the new batch can be easily added.

Like the DSTree, the DSTable is always kept up-to-date when the window slides. The corresponding mining algorithm first extracts relevant transactions from the DSTable. Then, the algorithm (i) constructs an FP-tree for the projected database of each of these 1-itemsets and (ii) recursively forms subsequent FP-trees for projected databases of frequent k-itemsets (where $k \geq 2$)—by traversing the paths of these FP-trees—to find all frequent patterns.

Because the DSTable also keeps w boundary values for each row (representing each of the m domain items) to facilitate easy insertion and deletion of contents in the DSTable when the window (of size w batches) slides, the DSTable needs a total of $m \times w$ boundary values. Moreover, each table entry is a pointer that indicates the location in terms of row name and column number of the table entry for the "next" item in the same transaction. When the data stream is sparse, only a few pointers need to be stored. However, when the stream is dense, many pointers need to be stored. Given a total of $|T|$ transactions in all batches within the current sliding window, there are potentially $m \times |T|$ pointers (where m is the number of domain items). Furthermore, during the mining process, multiple FP-trees need to be constructed and kept in memory (e.g., FP-trees for all $\{a\}$-, $\{a, c\}$- and $\{a, c, d\}$-projected databases are required to be kept in memory).

2.3. Stream mining with a data stream matrix (DSMatrix)

To avoid storing many pointers (i.e., potentially $m \times |T|$ pointers, where m is the number of domain items and $|T|$ is the number of transactions in all batches within the current sliding window) when the memory space is limited, *Data Stream Matrix* (*DSMatrix*)[3] can be used. Generally, a DSMatrix is a two-dimensional structure that captures the contents of transactions in all batches within the current sliding window by storing them on the disk. The DSMatrix is a binary matrix, which represents the presence of an item x in transaction t_i by a "1" in the matrix entry (t_i, x) and the absence of an item y from transaction t_j by a "0" in the matrix entry (t_j, y). With this binary representation of items in each transaction, each column in the DSMatrix captures a transaction. Each column in the DSMatrix can be considered as a bit vector.

When the window slides, the DSMatrix keeps track of any boundary between two batches so that transactions in the older batches can be easily removed and transactions in the newer batches can be easily added. Unlike the DSTable (in which boundaries may vary from one row representing an item to another row representing another item due to the potentially different number of items present), boundaries in DSMatrix are the same from one row to another because we put a binary value (0 or 1) for each transaction. Hence, the DSMatrix only keeps w boundary values (where $w \ll m \times w$) for the entire matrix, regardless how many domain items (m) are there. Moreover, as DSMatrix uses a bit vector to indicate the presence or absence of items in a transaction, the computation does not require us to keep track of the index of the last item in every row, thus incurring a lower computation cost. Given a total of $|T|$ transactions in all batches within the current sliding window, there are $|T|$ columns in our DSMatrix. Each column requires only m bits. In other words, the DSMatrix takes $m \times |T|$ bits (cf. potentially $64m \times |T|$ bits for dense data streams required by the DSTree).

3. Our indirect 2-step edge-based algorithm for mining frequent subgraphs from graph streams

After reviewing three structures (i.e., DSTree, DSTable, and DSMatrix) for mining data streams, let us present in this section our indirect 2-step edge-based algorithm for mining a specific class of streams—*streams of graph structured data* (i.e., *graph streams*). Like the general data streams, the graph streams can also be divided into batches. However, unlike the general data streams (in which each batch contains multiple transactions), each batch in the graph streams contains multiple graphs. Each graph $G = (V, E)$ consists of $|V|$ vertices and $|E|$ edges. See Example 1, which represents some insertions, deletions, and/or updates on the linkages among linked data in graphs or networks (e.g., linked documents in a semantic web, friendships in a social network, connections in sensor or road networks).

Table 1. A list of neighboring edges for the graph stream in Example 1 (Observation 1).

Edge	Neighboring edges	Edge	Neighboring edges	Edge	Neighboring edges
a	b, d, e and f	c	b, d, e and f	e	a, b, c and d
b	a, c, e and f	d	a, c, e and f	f	a, b, c and d

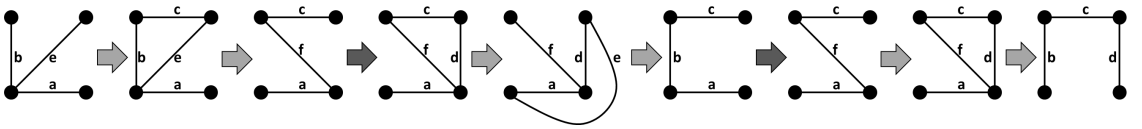

Fig. 1. A stream of graph structured data (Example 1).

Example 1. For illustrative purpose, let us consider the following stream of 9 graphs (as shown Fig. 1), where each graph $G_i = (V_i, E_i)$ consists of $|V_i| = 4$ vertices and $1 \leq |E_i| \leq 6$ edges labelled a, b, c, d, e & f (for $1 \leq i \leq 9$):

- $V_1 = V_2 = \ldots = V_8 = V_9$, whereas
- $E_1 = \{a, b, e\}$ at time T_1,
- $E_2 = \{a, b, c, e\}$ at time T_2,
- $E_3 = \{a, c, f\}$ at time T_3,
- $E_4 = \{a, c, d, f\}$ at time T_4,
- $E_5 = \{a, d, e, f\}$ at time T_5,
- $E_6 = \{a, b, c\}$ at time T_6,
- $E_7 = E_3 = \{a, c, f\}$ at time T_7,
- $E_8 = E_4 = \{a, c, d, f\}$ at time T_8, and
- $E_9 = \{b, c, d\}$ at time T_9.

As shown Fig. 1, edges in the above 9 graphs $E_1, E_2, \ldots, E_8$ & E_9 are arranged in the following way that they share some of 4 vertices:

- edges a, b & e share the same vertex;
- edges a, d & f share another vertex;
- edges b, c & f share the third of the four vertices; and
- edges c, d & e share the last vertex.

Observation 1. We observed from Example 1 that edge a (i) shares a vertex with edges b & e while (ii) sharing another vertex with edges d & f. In other words, b, d, e and f are neighboring edges of a. Similarly, we also observed the neighboring edges of the remaining five edges. All these neighboring edges are shown in Table 1.

With the above edge-based representation of graph streams, we can adapt a commonly used data stream processing model—namely, the *sliding window model*, which allows users to focus on graph-structured data in a fixed-size time window—to process these graph streams. Note that techniques presented in this paper can be easily adapted to the other stream processing models.

Recall from Section 2.3 that stream mining with the DSMatrix requires the least memory space when compared with the other two strcutures (DStree or DSTable). Thus, mining with the DSMatrix would be a good choice for handling high volumes of graph streams. To find frequent subgraphs from these graph streams, our indirect 2-step edge-based graph stream mining algorithm first constructs a DSMatrix to capture edges in those graphs within the current sliding window. When a new batch of graph streams flows in, the window slides. Graphs in the oldest batch in the sliding window are then removed from the DSMatrix so that graphs in this new batch can be added. In other words, the mining is "delayed" until it is needed. Once the DSMatrix is constructed, it is kept up-to-date on the disk. See Example 2.

Table 2. DSMatrices at the end of time T_3, T_6 and T_9 (Example 2).

(a) DSMatrix at the end of T_3

Row	Contents
	$E_1E_2E_3$
Edge a:	1 1 1;
Edge b:	1 1 0;
Edge c:	0 1 1;
Edge d:	0 0 0;
Edge e:	1 1 0;
Edge f:	0 0 1;
Boundary:	Column 3

(b) DSMatrix at the end of T_6

Row	Contents
	$E_1E_2E_3$ $E_4E_5E_6$
Edge a:	1 1 1; 1 1 1;
Edge b:	1 1 0; 0 0 1;
Edge c:	0 1 1; 1 0 1;
Edge d:	0 0 0; 1 1 0;
Edge e:	1 1 0; 0 1 0;
Edge f:	0 0 1; 1 1 0;
Boundaries:	Columns 3 & 6

(c) DSMatrix at the end of T_9

Row	Contents
	$E_4E_5E_6$ $E_7E_8E_9$
Edge a:	1 1 1; 1 1 0;
Edge b:	0 0 1; 0 0 1;
Edge c:	1 0 1; 1 1 1;
Edge d:	1 1 0; 0 1 1;
Edge e:	0 1 0; 0 0 0;
Edge f:	1 1 0; 1 1 0;
Boundaries:	Columns 3 & 6

Example 2. Consider the graph stream in Example 1. With a sliding window of size $w = 2$ batches (i.e., only two batches are kept) where each batch keeps graphs for three time instances, our indirect 2-step edge-based algorithm first constructs a DSMatrix to capture edges in those graphs within the current sliding window. Consequently, at the end of time T_3, the DSMatrix keeps the three graphs with edges $E_1 = \{a, b, e\}$, $E_2 = \{a, b, c, e\}$ and $E_3 = \{a, c, f\}$ from the first batch B_1. See Table 2(a).

At the end of time T_6, the DSMatrix keeps (i) these three graphs with edges $E_1 = \{a, b, e\}$, $E_2 = \{a, b, c, e\}$ and $E_3 = \{a, c, f\}$ from the first batch B_1; and (ii) adds three new graphs with edges $E_4 = \{a, c, d, f\}$, $E_5 = \{a, d, e, f\}$ and $E_6 = \{a, b, c\}$ from the second batch B_2. In addition, the DSMatrix also keeps track of the global boundary information, which is applicable for all rows/graphs. See Table 2(b).

When the third batch (batch B_3) of graph streams flows in, the window slides. At the end of time T_9, the DSMatrix uses the boundary information to (i) remove all columns up to Column 3 (i.e., those three graphs with edges $E_1 = \{a, b, e\}$, $E_2 = \{a, b, c, e\}$ and $E_3 = \{a, c, f\}$ belonging to the first batch B_1) and (ii) keeps all graphs in Column (3+1) to Column 6 (or more precisely, shifts all columns from Columns 4–6 to Columns 1–3). In other words, the DSMatrix keeps the three graphs with edges $E_4 = \{a, c, d, f\}$, $E_5 = \{a, d, e, f\}$ and $E_6 = \{a, b, c\}$ from the second batch B_2. Moreover, the DSMatrix appends three new graphs with edges $E_7 = E_3 = \{a, c, f\}$, $E_8 = E_4 = \{a, c, d, f\}$ and $E_9 = \{b, c, d\}$ from the third batch B_3. Again, the DSMatrix keeps track of the global boundary information, which is applicable for all rows/graphs. See Table 2(c).

3.1. Step 1 of our 2-step algorithm: discovering all frequent edges

With the bitwise representation of graph streams in the DSMatrix, it is logical to mine frequent subgraphs *vertically*. Specifically, our edge-based algorithm examines each row (representing an edge). The *row sum* (i.e., total number of 1s) gives the frequency of the edge represented by that row. Any edge with row sum (i.e., frequency) ≥ a user-specified threshold *minsup* is considered *frequent*.

Example 3. Recall from Example 2 that important contents of the graph stream (e.g., contents of the $w = 2$ batches of graphs—more specifically, edges of these graphs) shown in Example 1 are captured by the DSMatrix. Let *minsup* be set to 2. When applying our 2-step edge-based algorithm to this graph stream, the algorithm counts the row sum of the DSMatrix shown in Table 2(b) and discovers that all edges are frequent at the end of time T_6 because the frequencies of edges a, b, c, d, e and f are 6, 3, 4, 2, 3 and 3, respectively. A frequency of 6 for edge a means that edge a appears in all 6 graphs $G_1, G_2, \ldots, G_6$.

Similarly, by counting the row sum of the DSMatrix shown in Table 2(c), our 2-step edge-based algorithm finds that edge e is no longer frequent. In other words, all except edge e are frequent at the end of time T_9 because the frequencies of edges a, b, c, d, e and f are 5, 2, 5, 4, 1 and 4, respectively. Note that the frequency of edge a is dropped from 6 (appearing in all 6 graphs $G_1, G_2, \ldots, G_6$) to 5 (appearing in only 5 graphs $G_4, G_5, \ldots, G_8$).

Once the frequent singleton edges are found, we *intersect the bit vectors* for two edges. If the row sum of the resulting intersection ≥ a user-specified threshold *minsup*, then we find a frequent edge-pair (i.e., a collection of

2 frequent edges). We then find edge-triplets (i.e., collections of 3 edges) by intersecting two edge-pairs that *share a common edge*. Afterwards, we apply a similar procedure to find every collection of k edges (for $k \geq 4$).

To speed up this mining step, our 2-step edge-based algorithm intersects the bit vectors for two *frequent* edges. If the row sum of the resulting intersection $\geq$ a user-specified threshold *minsup*, then we find a frequent edge-pair. Our algorithm then finds edge-triplets (i.e., collections of 3 edges) by intersecting two *frequent* edge-pairs that share a common edge. The same procedure is repeated recursively by intersecting those *frequent* intersection results that share some common edges to find every collection of k edges (for $k \geq 4$). See Example 4.

Example 4. Continue with Example 3. After finding five frequent singleton edges a, b, c, d and f at the end of time T_9, our 2-step edge-based algorithm intersects the bit vector of frequent singleton edge a with any one of the remaining four bit vectors (for frequent singleton edges b, c, d and f) to find three frequent edge-pairs $\{a, c\}, \{a, d\}$ and $\{a, f\}$ with frequencies 4, 3 and 4, respectively, because (i) the intersection of $\overrightarrow{a}$=111110 and $\overrightarrow{c}$=101111 gives bit vector $\overrightarrow{ac}$=101110, (ii) the intersection of $\overrightarrow{a}$ and $\overrightarrow{d}$=110011 gives bit vector $\overrightarrow{ad}$=110010, and (iii) the intersection of $\overrightarrow{a}$ and $\overrightarrow{f}$=110110 gives bit vector $\overrightarrow{af}$=110110. Note that the intersection of $\overrightarrow{d}$ and $\overrightarrow{b}$=001001 gives a bit vector 001000 with a row sum < *minsup*.

Next, the algorithm intersects (i) $\overrightarrow{ac}$ with $\overrightarrow{ad}$ to get $\overrightarrow{acd}$=100010, (ii) $\overrightarrow{ac}$ with $\overrightarrow{af}$ to get $\overrightarrow{acf}$=100110, and (iii) $\overrightarrow{ad}$ with $\overrightarrow{af}$ to get $\overrightarrow{adf}$=110010. Hence, it finds frequent three edge-triplets $\{a, c, d\}, \{a, c, f\}$ and $\{a, d, f\}$.

The algorithm also intersects $\overrightarrow{acd}$ with $\overrightarrow{acf}$ to find a frequent edge-quadruplet $\{a, c, d, f\}$ with $\overrightarrow{acdf}$=100010. So far, the algorithm has found all 1+3+3+1 = 8 collections of frequent edges containing a: $\{a\}$; $\{a, c\}, \{a, d\}, \{a, f\}$; $\{a, c, d\}, \{a, c, f\}, \{a, d, f\}$; and $\{a, c, d, f\}$.

Afterwards, our algorithm applies similar steps with the bit vectors for other edges. For instance, it intersects $\overrightarrow{b}$ with $\overrightarrow{c}$, $\overrightarrow{d}$ and $\overrightarrow{f}$, and finds out that—among them—only $\{b, c\}$ is frequent with frequency 2. The algorithm also intersects $\overrightarrow{c}$ with $\overrightarrow{d}$ and $\overrightarrow{f}$ to find frequent two edge-pairs $\{c, d\}$ and $\{c, f\}$, each having frequency 3 as $\overrightarrow{cd}$=100011 and $\overrightarrow{cf}$=100110. It also finds a frequent edge-triplet $\{c, d, f\}$ by intersecting $\overrightarrow{cd}$ and $\overrightarrow{cf}$. Finally, it intersects $\overrightarrow{d}$ with $\overrightarrow{f}$ to find a frequent edge-pair $\{d, f\}$ with frequency 3. Consequently, our algorithm finds a total of 17 collections of frequent k edges (where $1 \leq k \leq 4$), which include 8 collections (containing a), 1+1=2 collections (containing b but not a), 1+2+1=4 collections (containing c but not a or b), 1+1=2 collections (containing d but not a, b or c), and 1 collection (containing only f).

3.2. Step 2 of our 2-step algorithm: pruning disjoint frequent edges

Once Step 1 of our 2-step algorithm has found collections of all frequent edges—which include *connected* edges such as $\{a, d\}$ as well as *disjoint* edges such as $\{a, c\}$, Step 2 of our algorithm applies a post-processing step to check every frequent edge to prune those disjoint ones. Based on Observation 1, if two edges are connected (by sharing a vertex), then one edge is a neighboring edge of another. Two edges are *disjoint* if they are not neighboring edges (i.e., do not share any common vertex). For instance, we check and keep $\{a, d\}$ because d is a neighboring edge of a. However, we check and prune away $\{a, c\}$ because c is not a neighboring edge of a. See Example 5.

Example 5. Recall from Example 4 that Step 1 of our 2-step algorithm found 17 collections of frequent edges, which may include some disjoint edges (i.e., false positives). So, Step 2 checks these collections to determine if any are disjoint. Specifically, upon checking, our algorithm finds that frequent edge-pairs $\{a, d\}$ and $\{a, f\}$ are connected because both d and f are neighbouring edges of a. However, as c is not a neighbouring edge of a, frequent edge-pair $\{a, c\}$ is disjoint and can thus be pruned.

Similarly, frequent edge-pair $\{b, c\}$ is also connected (because c is a neighbouring edge of b); frequent edge-pairs $\{c, d\}$ and $\{c, f\}$ are connected (because both d and f are neighbouring edges of c). Finally, frequent edge-pair $\{d, f\}$ is also connected (because f is a neighbouring edge of d).

Note that the four frequent edge-triplets $\{a, c, d\}, \{a, c, f\}, \{a, d, f\}$ and $\{c, d, f\}$, as well as the one frequent edge-quadruplet $\{a, c, d, f\}$, found in Step 1 are all connected.

To summarize, among the 17 collections of frequent edges, Step 2 of our 2-step algorithm prunes a disjoint edge-pair $\{a, c\}$ in this illustrative example. The algorithm returns $17 - 1 = 16$ frequent *connected* subgraphs. More pruning is expected for bigger graphs in the stream.

Observation 2. We observed from Example 5 that the 2-step algorithm does not need to check all 17 collections of frequent k edges to determine which disjoint collections need to be pruned. Specifically, as the algorithm finds a collection of k+1 edges by intersecting two collections of k frequent edges that *share a common edge* for $k \geq 2$ (e.g., intersecting two frequent edge-pairs to get an edge-triplet), all the resulting collections of k+1 edges are guaranteed to be connected. Hence, the algorithm only needs to check 3+1+2+1 = 7 edge-pairs.

4. Our direct 1-step edge-based algorithm for mining frequent subgraphs from graph streams

The previous section shows how our 2-step algorithm indirectly mines frequent subgraphs from graph streams by discovering all collections of frequent edges in Step 1 and then pruning collections of disjoint edges in Step 2. Such an indirect mining approach may incur a lot of time and effort in discovering all collections of frequent edges—including many disjoint edges, which are then pruned.

To deal with this issue, we present our 1-step algorithm that directly mines frequent subgraphs in this section. This algorithm first mines frequent singleton edges in the same way as in our 2-step algorithm. Then, unlike our 2-step algorithm, we intersect the bit vectors for two *connected* edges—based on the neighborhood information—to find *connected* edge-pairs (i.e., collections of 2 *connected* edges). Afterwards, we intersect two collections of k frequent connected edges that *share a common edge* to get a collection of k+1 connected edges. Since the two edges for intersection share a common edge, they are guaranteed to be connected.

Example 6. Revisit Examples 3–5. With *minsup*=2, our 1-step edge-based algorithm directly mines frequent subgraphs as follows. It first discovers 5 frequent singleton edges a, b, c, d and f. These are edges with row sums in the DSMatrix in Table 2(c) $\geq$ *minsup*.

The algorithm then intersects bit vectors of *connected* edges such as (i) $\overrightarrow{a}$ with $\overrightarrow{b}$, $\overrightarrow{d}$ and $\overrightarrow{f}$ to get $\overrightarrow{ab}$, $\overrightarrow{ad}$ and $\overrightarrow{af}$; (ii) $\overrightarrow{b}$ with $\overrightarrow{c}$ and $\overrightarrow{f}$ to get $\overrightarrow{bc}$ and $\overrightarrow{bf}$; (iii) $\overrightarrow{c}$ with $\overrightarrow{d}$ and $\overrightarrow{f}$ to get $\overrightarrow{cd}$ and $\overrightarrow{cf}$; as well as (iv) $\overrightarrow{d}$ with $\overrightarrow{f}$ to get $\overrightarrow{df}$. Among them, $\overrightarrow{bf}$ is infrequent, and thus edge-pair $\{b, f\}$ is not returned to users. Moreover, when compared with our 2-step algorithm, this 1-step algorithm saves some computation by not intersecting $\overrightarrow{a}$ with $\overrightarrow{c}$ because edges a and c are disjoint. Similarly, our 1-step algorithm also saves some computation by not needing to count frequency for the infrequent *disjoint* edge-pair $\{b, d\}$ because it does not even intersect $\overrightarrow{b}$ with $\overrightarrow{d}$.

Afterwards, the algorithm intersects two collections of k frequent connected edges that *share a common edge* to get a collection of k+1 connected edges. For instance, it intersects $\overrightarrow{ad}$ with $\overrightarrow{af}$ to get $\overrightarrow{adf}$, $\overrightarrow{ad}$ with $\overrightarrow{cd}$ to get $\overrightarrow{acd}$, $\overrightarrow{af}$ with $\overrightarrow{cf}$ to get $\overrightarrow{acf}$, $\overrightarrow{acd}$ with $\overrightarrow{acf}$ to get $\overrightarrow{acdf}$, and $\overrightarrow{cd}$ with $\overrightarrow{cf}$ to get $\overrightarrow{cdf}$. Consequently, this 1-step algorithm directly discovers a total of 16 frequent subgraphs (i.e., 16 collections of frequent *connected* edges): $\{a\}$; $\{a, d\}$, $\{a, f\}$; $\{a, c, d\}$, $\{a, c, f\}$, $\{a, d, f\}$; $\{a, c, d, f\}$; $\{b\}$; $\{b, c\}$; $\{c\}$; $\{c, d\}$, $\{c, f\}$; $\{c, d, f\}$; $\{d\}$; $\{d, f\}$; and $\{f\}$.

5. Evaluation

To evaluate our two edge-based frequent subgraph mining algorithms, we generated graph streams by using random graph models via a Java-based generator with various model parameters (e.g., topology, average fan-out of nodes, edge centrality, etc.) and derived edges from the graph models. In addition, we also used many different datasets including IBM synthetic datasets, real-life datasets (e.g., connect4) from the UC Irvine Machine Learning Depository as well as those from the Frequent Itemset Mining Implementation (FIMI) Dataset Repository. For example, connect4 is a dense data set containing 67,557 records. Each record represents a graph of legal 8-ply positions in the game of connect 4. All experiments were run in a time-sharing environment on a 1 GHz machine. We set each batch to be 6K records and the window size w=5 batches. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os.

We first measured the accuracy of mining with the following structures: (i) DSTree[23], (ii) DSTable[5], and (iii) DSMatrix. Experimental results show that our two algorithms (which both use the DSMatrix) gave the same mining results as existing algorithms that use DSTree and DSTable.

We also measured the space efficiency. Experimental results show that mining with the DSTree stored one global DSTree and multiple local FP-trees in main memory, and thus took the largest main memory space. Mining with

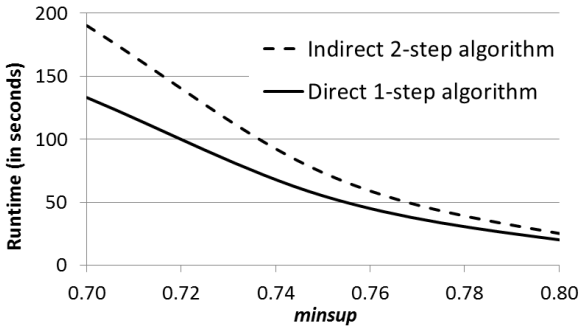

Fig. 2. Evaluation results.

the DSTable and DSMatrix required less memory because the DSTable and DSMatrix were kept on disk. Our two algorithms (Sections 3 and 4) required the least amount of memory space because they both work with bit vectors.

In addition, we measured the time efficiency. Our mining algorithms (Sections 3 and 4) also required the shortest runtime when compared with existing algorithms that mine with the DSTree or DSTable because ours work with bitwise and set intersection operators. Between our two algorithms, as expected, the indirect 2-step algorithm required longer runtime than the direct 1-step algorithm because the latter mines frequent connected subgraphs directly. See Fig. 2.

Furthermore, we performed some additional experiments (e.g., evaluating the effect of *minsup*). Results on Fig. 2 show that the runtime decreased when *minsup* increased. The results on varying the number of batches in the graph stream show the scalability of our two edge-based mining algorithms.

6. Conclusions

In the current era of Big data, high volumes of valuable data can be generated at a high velocity from high-varieties of data sources in various real-life applications ranging from sensor networks to social networks, from bio-informatics to chemical informatics. In addition, Big data are also available in business, education, engineering, finance, healthcare, scientific, telecommunication, and transportation domains. A collection of these data can be viewed as a big dynamic graph structure. Embedded in them are implicit, previously unknown, and potentially useful knowledge. Consequently, efficient knowledge discovery algorithms for mining frequent subgraphs from these dynamic streaming graph structured data are in demand. On the one hand, some existing algorithms discover collections of frequently co-occurring edges, which may be disjoint. On the other hand, some other existing algorithms discover frequent subgraphs by requiring very large memory space. With high volumes of Big data, available memory space may be limited. To discover collections of frequently co-occurring *connected* edges, we presented in this paper two time-efficient and space-efficient edge-based algorithms that mine frequent subgraphs from graph streams. Our indirect 2-step algorithm first discovers all frequent edges and then prunes disjoint edges at a post-processing step, whereas our direct 1-step algorithm pushes the pruning step early inside in the mining process for discovering all frequent connected edges. Experimental results show the accuracy and efficiency of both edge-based algorithms in mining frequent subgraphs from graph streams.

Acknowledgements

This project is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Manitoba.

References

1. Aggarwal CC, Li Y, Yu PS, Jin R. On dense pattern mining in graph streams. *PVLDB* 2010; **3**(1–2):975–984.
2. Bifet A, Holmes G, Pfahringer B, Gavaldà R. Mining frequent closed graphs on evolving data streams. In: *Proceedings of the ACM KDD 2011*. ACM; 2011, p. 591–599.
3. Braun P, Cameron JJ, Cuzzocrea A, Jiang F, Leung CK. Effectively and efficiently mining frequent patterns from dense graph streams on disk. *Procedia Computer Science* 2014; **35**:338–347.
4. Braun P, Cuzzocrea A, Leung CK, MacKinnon RK, Tanbeer SK. A tree-based algorithm for mining diverse social entities. *Procedia Computer Science* 2014; **35**:223–232.
5. Cameron JJ, Cuzzocrea A, Leung CK. Stream mining of frequent sets with limited memory. In: *Proceedings of the ACM SAC 2013*. ACM; 2013, p. 173–175.
6. Chi L, Li B, Zhu X. Fast graph stream classification using discriminative clique hashing. In: *Proceedings of the PAKDD 2013, Part I*. Springer; 2013, p. 225–236.
7. Cuzzocrea A. CAMS: OLAPing multidimensional data streams efficiently. In: *Proceedings of the DaWaK 2009*. Springer; 2009, p. 48–62.
8. Cuzzocrea A, Chakravarthy S. Event-based lossy compression for effective and efficient OLAP over data streams. *DKE* 2010; **69**(7):678–708
9. Cuzzocrea A, Furfaro F, Mazzeo GM, Saccà D. A grid framework for approximate aggregate query answering on summarized sensor network readings. In: *Proceedings of the OTM Workshops 2004*. Springer; 2004, p. 144–153.
10. Cuzzocrea A, Jiang F, Leung CK. Frequent subgraph mining from streams of linked graph structured data. In: *Proceedings of the EDBT/ICDT Workshops 2015*. CEUR-WS.org; 2015, p. 237–244.
11. Czarnowski I, Jedrzejowicz P. Ensemble classifier for mining data streams. *Procedia Computer Science* 2014; **35**:397–406.
12. Fariha A, Ahmed CF, Leung CK, Abdullah SM, Cao L. Mining frequent patterns from human interactions in meetings using directed acyclic graphs. In *Proceedings of the PAKDD 2013, Part I*. Springer; 2013, p. 38–49.
13. Giannella C, Han J, Pei J, Yan X, Yu PS. Mining frequent patterns in data streams at multiple time granularities. In: Kargupta H, Joshi A, Sivakumar K, Yesha Y, editors. *Data Mining: Next Generation Challenges and Future Directions*. AAAI Press; 2004, chap. 6.
14. Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In: *Proceedings of the ACM SIGMOD 2000*. ACM; 2000, p. 1–12.
15. Jiang F, Leung CK. Mining interesting "following" patterns from social networks. In: *Proceedings of the DaWaK 2014*. Springer; 2014, p. 308–319.
16. Jiang F, Leung CK, Liu D, Peddle AM. Discovery of really popular friends from social networks. In: *Proceedings of the IEEE BDCloud 2014*. IEEE Computer Society; 2014, p. 342–349.
17. Jiang F, Leung CK, MacKinnon RK. BigSAM: mining interesting patterns from probabilistic databases of uncertain big data. In: *Proceedings of the PAKDD Workshops 2014*. Springer; 2014, p. 780–792.
18. Leung CK. Mining frequent itemsets from probabilistic datasets. In: *Proceedings of the EDB 2013*. KIISE; 2013, p. 137–148.
19. Leung CK, Carmichael CL. Exploring social networks: a frequent pattern visualization approach. In: *Proceedings of the IEEE SocialCom 2010*. IEEE Computer Society; 2010, p. 419–424.
20. Leung CK, Cuzzocrea A, Jiang F. Discovering frequent patterns from uncertain data streams with time-fading and landmark models. *LNCS TLDKS* 2013; **8**:174–196.
21. Leung CK, Jiang F. A data science solution for mining interesting patterns from uncertain big data. In: *Proceedings of the IEEE BDCloud 2014*. IEEE Computer Society; 2014, p. 235–242.
22. Leung CK, Joseph KW. Sports data mining: predicting results for the college football games. *Procedia Computer Science* 2014; **35**:710–719.
23. Leung CK, Khan QI. DSTree: a tree structure for the mining of frequent sets from data streams. In: *Proceedings of the IEEE ICDM 2006*. IEEE Computer Society; 2006, p. 928–932.
24. Leung CK, MacKinnon RK. BLIMP: a compact tree structure for uncertain frequent pattern mining. In: *Proceedings of the DaWaK 2014*. Springer; 2014, p. 115–123.
25. Leung CK, MacKinnon RK, Tanbeer SK. Tightening upper bounds to the expected support for uncertain frequent pattern mining. *Procedia Computer Science* 2014; **35**:328–337.
26. MacKinnon RK, Strauss TD, Leung CK. DISC: efficient uncertain frequent pattern mining with tightened upper bounds. In: *Proceedings of the IEEE ICDM Workshops 2014*. IEEE Computer Society; 2014, p. 1038–1045.
27. Papapetrou O, Garofalakis M, Deligiannakis A. Sketch-based querying of distributed sliding-window data streams. *PVLDB* 2012; **5**(10):992–1003.
28. Ríos SA, Videla-Cavieres IF. Generating groups of products using graph mining techniques. *Procedia Computer Science* 2014; **35**:730–738.
29. Tanbeer SK, Jiang F, Leung CK, MacKinnon RK, Medina IJM. Finding groups of friends who are significant across multiple domains in social networks. In: *Proceedings of the CASoN 2013*. IEEE Computer Society; 2013, p. 21–26.
30. Tanbeer SK, Leung CK, Cameron JJ. Interactive mining of strong friends from social networks and its applications in e-commerce. *Journal of Organizational Computing and Electronic Commerce* 2014; **24**(2–3):157–173.
31. Tirthapura S, Woodruff DP. A general method for estimating correlated aggregates over a data stream. In: *Proceedings of the IEEE ICDE 2012*. IEEE Computer Society; 2012, p. 162–173.
32. Valari E, Kontaki M, Papadopoulos AN. Discovery of top-k dense subgraphs in dynamic graph collections. In: *Proceedings of the SSDBM 2012*. Springer; 2012, p. 213–230.